

# Energy Optimization of Source Code Guided by a Fine-Grained Energy Model

Xueliang Li      John P. Gallagher  
 {xueliang, jpg}@ruc.dk  
 Roskilde University  
 Denmark

## ABSTRACT

Energy efficiency can have a significant influence on user experience of mobile devices such as smartphones and tablets. Software optimization plays an important role in saving energy; however, effective energy optimization relies mainly on developers, since compiler-based energy optimizations are limited. In this paper, we propose an energy-aware programming approach, guided by an operation-based source-level energy model, which allows the programmer to understand the energy usage of the code and then apply targeted refactoring to save energy. To the best of our knowledge, our work is the first that achieves this for a high-level language such as Java. This approach can be applied at the end of the software engineering implementation phase in order to avoid distracting developers from guaranteeing the correctness of software. In a case study, experimental evaluation shows that our approach is able to save from 6.4% to 50.2% of the overall energy consumption in various application scenarios.

## 1. INTRODUCTION

Smartphones are one of the most important inventions of modern society. In February 2015, the penetration of smartphones was about 75% in the U.S. [4]. This figure is still growing. With the improvement of hardware processing capability and software development environments, the smartphone is no longer just a handset to make phone calls, but also lets the user play entertaining games, watch movies, browse web pages, and so on. On the other hand, users are often frustrated by limited battery capacity – applications running in parallel could easily drain a fully-charged battery within 24 hours.

Software optimization by the compiler achieves very little energy-saving for mobile devices, since besides energy efficiency, the compiler for the mobile device has to consider many other important factors, such as limited memory usage and quick responses to user interactions. The Android platform, for instance, employs the Just-In-Time (JIT) compiler [6], also known as the dynamic compiler. Its optimization window is generally as small as one or two basic blocks in order to use less memory and speed up delivery of performance boost. However, the small window largely restricts the space of energy-saving strategies. More powerful code

refactoring is needed, but this is beyond the scope of compilers and relies more on developers.

Unfortunately, current software development is performed in an energy-oblivious manner. Throughout the engineering life cycle, most developers and designers are blind to the energy usage of code written by themselves. However, developers are desperate for knowledge on energy-aware programming techniques. In the most popular software development forum STACKOVERFLOW [40], energy-related questions are marked as favorites 3.89 more often than the average questions [34]. Furthermore, among the energy-related questions, code-design-related ones are prominent. Moreover, it has been estimated that energy-saving by a factor of as much as three to five could be achieved solely by software optimization [12]. To realize this, the first step is to analyze the energy attributes of source code at different levels of granularity and from different points of view.

In order to expose energy attributes of code, energy modeling of code is needed to bridge the gap between high-level source code and low-level hardware, where energy is consumed. However, traditional bottom-to-top modeling techniques [8, 36, 42, 44] face obstacles when the software stack of the system consists of a number of abstract layers. On the Android platform, for instance, the source code is in Java and then translated to Java byte-code, further to Dalvik [3] byte-code, native code and machine code and finally executes on the processors and dissipates energy. Consequently, the modeling task has to describe the links between all the layers.

Instead of building a software energy model layer by layer, another approach to acquiring software-level energy information is to use the hardware readings, like CPU state residency, CPU utilization, L1/L2 Cache misses and battery trace, as predictors of software energy use [11, 33, 45, 46]. However, they are only capable of obtaining energy information at a coarse level of granularity such as methods or even applications. Two pieces of work [14, 20] result in source-line energy information. The former requires low-level energy profiles. The latter employs an accurate measurement to obtain the energy dissipation of source lines.

The energy information on blocks or more coarse-grained units could identify the hot spots in the code, but it gives

few clues about how to make changes to improve the code. The source line is also not an appropriate level of granularity to provide energy information. For instance, the header of for loop contains three segments which are *initialization*, *boolean* and *update* in the same source line, but usually have distinct numbers of executions.

Our latest work constructs a source-level energy model based on "energy operations", which is more fine-grained and gives more valuable information for code optimization.

By "source-level" we mean that the energy costs of running a program are all attributed to source code constructs, despite the fact that much of the energy consumed is actually accounted for by things outside the source code such as the operating system. Thus the model is bound to be an approximation, yet as our results show, it is precise enough to give useful information.

Compared with coarse-grained techniques, there are some advantages of the operation-based model in guiding energy-aware programming techniques:

- The energy operations are basic units that constitute the energy consumption of the entire application. Thus using the energy estimate of operations, developers can assess the effects of code changes on the energy consumption of code.
- It provides more valuable information for refactoring. For example, the experiment shows that method invocation is one of the most expensive operations, suggesting that in some cases we may inline some thin methods, at the cost of losing the integrity of the structure of code.

In this paper, we propose an energy-aware programming approach guided by a fine-grained energy model of source code. The summary procedure of the approach is the following:

- We build an operation-based source-level energy model, which is achieved by analyzing the data produced in a range of well-designed execution cases.
- We perform energy accounting based on the model, at operation and block level to capture the key energy characteristics of the code.
- We focus efforts on the most costly blocks, where we refactor the code to remove, reduce or replace the expensive operations, while maintaining its logical consistency with the original code.

Our target platform is an Android development board with two ARM quad-core CPUs, and the source code in our study is a game engine used in games, demos and other interactive applications. We evaluate the approach in three game scenarios, and the experimental result shows that it can save energy consumption from 6.4% up to 50.2% depending on different scenarios.

Table 1: Examples of Energy Operations

Operation	Identified where:
Method Invocation	<i>one method is called</i>
Parameter_Object	<i>Object is one parameter of the method</i>
Return_Object	<i>the method returns an Object</i>
Addition_int_int	<i>addition's operands are integers</i>
Multi_float_float	<i>multiplication's operands are floats</i>
Increment	<i>symbol "++" appears in code</i>
And	<i>symbol "&amp;&amp;" appears in code</i>
Less_int_float	<i>"&lt;"s operands are integer and float</i>
Equal_Object_null	<i>"=="s operands are Object and null</i>
Declaration_int	<i>one integer is declared</i>
Assign_Object_null	<i>assignment's operands are Object and null</i>
Assign_char[]_char[]	<i>assignment's operands are arrays of chars</i>
Array Reference	<i>one array element is referred</i>
Block Goto	<i>the code execution goes to a new block</i>

The generality of the approach goes beyond the boundaries of the case study described here. Firstly, the energy-aware programming approach can be used in developing the large class of applications which are based on the game-engine, comprising many interactive applications with rich user interfaces. Secondly, the approach is applicable to all kinds of applications. The choice of energy operations is dependent only on the Java source language; the techniques for designing test cases, regression analysis and code optimization can be applied to other application domains.

In the rest of this paper, we begin with the identification of energy operations in Section 2. In Sections 3 and 4, we briefly summarize (to make the paper self-contained) the setup and construction of the energy model. Based on the model we are able to capture energy characteristics and optimize the source code in three different scenarios, Click & Move, Orbit and Waves, as seen in Sections 5, 6 and 7 respectively.

## 2. BASIC ENERGY OPERATIONS

Energy operations are identified directly from source code. The enumeration of the operations is inspired by Java semantics [7], which specifies the operational meaning, or behavior, of the Java language, which is the target language in the experiment. We intuitively identify semantic operations that perform operations on the state and may be energy-consuming, and let them be our energy operations. Ones that have little or no energy effect will automatically be identified by the regression analysis in the later stage of the analysis. Table 1 lists 14 representative operations out of a total of 120 in the experiment. They include arithmetic calculations like *Multi\_float\_float*, *Addition\_int\_int*, in which operands types are explicit, as well as *Increment* whose operand is implicitly an integer. Boolean operations and comparisons, such as *And*, *Less\_int\_float* and *Equal\_Object\_null* also form one major part. *Method Invocation* and *Block Goto* are important for the control flow which plays a key role in the exe-

Table 2: Examples of Library Functions

Class	Function
ArrayList	<i>add, get, size, isEmpty, remove</i> <i>glBindTexture, glDisableClientState</i> <i>glDrawElements, glEnableClientState</i>
GL10	<i>glMultMatrixf, glTexCoordPointer</i> <i>glPopMatrix, glPushMatrix</i> <i>glTexParameterx, glVertexPointer</i>
Math	<i>max, pow, sqrt, random</i>
FloatBuffer	<i>position, put</i>

cution of the code. Assignments and *Array Reference* will unexpectedly take a significant amount of the application’s energy consumption, as will be shown in Section 5.1.

The game engine application, like many others, also employs a diversity of library functions. Unlike the normal source code which is interpreted at run-time, the key part of library code has been compiled into native code before execution and some part may be already written in different languages and at lower levels of the software stack. On the other hand, usually a limited number (67 in the experiment) of library functions are frequently called in one application. So we treat them as basic modeling units. The examples of highly-used library functions in the experiment are shown in Table 2. For instance, the functions in the class of *GL10* are responsible for graphic computing.

### 3. EXPERIMENTAL SETUP

In this section and the next, we summarize the construction of the energy model, including the setup of the target device and the design principles of the execution cases. Further details on these can be found in our recent work<sup>1</sup>.

#### 3.1 Target Device

Experimental target: we employ an Odroid-XU+E development board [30] as the target device. It possesses two ARM quad-core CPUs, which are Cortex-A15 with 2.0 GHz clock rate and Cortex-A7 with 1.5 GHz. The eight cores are logically grouped into four pairs. Each pair consists of one big and one small core. So from the operating system’s point of view there are four logic cores. In our experiment, we turn off the small cores and run workload on big cores at a fixed clock frequency of 1.1 GHz. We do this in order to remove the influence of voltage, clock rate and CPU performance on the power usage. Odroid-XU+E has a built-in power monitoring tool to measure the voltage and current of CPUs with a frequency of 30 Hz.

#### 3.2 Target Source Code

The target source code is the Cocos2d-Android [2] game engine, a framework for building games, demos and other

<sup>1</sup>To preserve anonymity in the review process, the citation is hidden

interactive applications such as virtual reality. It also implements a fully-featured physics engine. Games are increasingly popular on mobile phones and include more and more fancy and energy-consuming features, requiring high CPU performance. This paper demonstrates the energy modeling, accounting and improvement for the source code of the game engine, and evaluates the improvement in three game scenarios.

### 3.3 Design of Execution Cases

The execution cases whose energy usage is measured and analyzed represent typical sequences of actions during game, including user inputs. We focus on three scenarios which are Click & Move, Orbit and Waves.

In the Click & Move scenario, the sprite (the character in the game) moves to the position where the tap occurs. In the Orbit scenario, the sprite together with the grid background spins in the three-dimension space. In the Waves scenario, the sprite scales up and down, meanwhile the grid background waves like flow. In both the Orbit and Waves scenarios, the animation will restart from the starting point whenever and wherever the tap occurs.

To simulate the game scenarios under different sequences of user inputs, we script with the Android Debug Bridge [1] (ADB), a command line tool connecting the target device to the host, to automatically feed the input sequences to the target device.

In order to obtain a more varied set of execution cases and thus a more precise model, we vary the executions of individual basic blocks in the code. This is achieved by systematically removing a set of blocks for each execution case, using the control flow graph extracted using the Soot tool [38]. We ensure that each block could be removed in some execution case. Thus an execution case is made up of one user input sequence and one set of basic blocks.

### 4. MODEL CONSTRUCTION

The entire energy use is composed of three parts: the cost of energy operations, the cost of library functions and the idle cost. The aimed model is formalized in Equation (1):

$$\begin{aligned}
 E = & \sum_{op_i \in EnergyOps} Cost_{op_i} \cdot N_e(op_i) \\
 & + \sum_{func_i \in LibFuncs} Cost_{func_i} \cdot N_e(func_i) + Idle\ Cost
 \end{aligned} \tag{1}$$

The cost of energy operations is the sum of  $Cost_{op_i} \cdot N_e(op_i)$  (the cost of one operation multiplied by the number of its executions), where  $op_i \in EnergyOps$ ; *EnergyOps* is the set containing all the operations. The cost of library functions is the sum of  $Cost_{func_i} \cdot N_e(func_i)$  (the cost of one library function multiplied by the number of its executions), where  $func_i \in LibFuncs$ ; *LibFuncs* is the set of library functions. The *Idle Cost* is the energy consumption of the device when

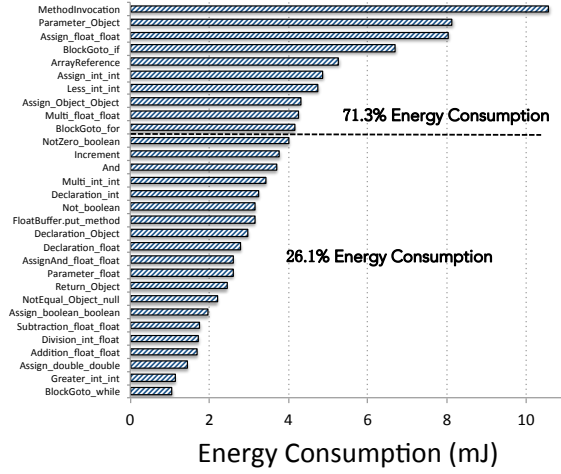


Figure 1: The top 30 energy consuming operations in Click & Move scenario.

running no application, simply the Android system. The lengths of case sessions are diversified due to input sequences, so the *Idle Cost* is different for each execution case. The model construction is based on regression analysis, finding out the correlation between energy operations and their costs from the data obtained in the execution cases.

## 5. THE CLICK & MOVE SCENARIO

In this section, we begin with energy accounting at operation and block level for the Click & Move scenario, after which we improve the most costly blocks focusing on the most expensive operations. We apply a similar approach in the other scenarios Orbit and Waves in Section 6 and Section 7; however for those cases we will only briefly summarize energy accounting and focus on the code improvements.

### 5.1 Energy Accounting

The energy model of app source code based on energy operations facilitates comprehensive energy accounting from operation-level up to source-level. In this section, we will see the rank of the most expensive operations, and the contributions of different operations to the energy consumption of each block.

#### Operation Level.

Figure 1 shows the top 30 energy consuming operations in the model, ranked by their single-execution energy costs. The line marked "71.3% Energy Consumption" indicates the percentage of the energy cost of the execution cases for the Click & Move scenario contributed by the top 10 operations. Similarly, "26.1% Energy Consumption" shows the contribution to the total cost of operations from 11th to 30th. We can see that the energy usage of the code is largely determined (97.4%) by a relatively small number of operations. This is due to the fact that these operations are frequently

used and expensive in themselves.

It might be supposed that the sophisticated arithmetic operations, such as multiplications and divisions, should be the most costly. However, the result shows that *Method Invocation* ranks the highest. This is due to a sequence of complex processes to fulfill *Method Invocation*, for example, most of the method calls in Java are virtual invocations which are dispatched on the type of the object at run-time and always implicitly passed a "this" reference as their first parameter, not to mention other operations such as storing the return address and managing the stack frame.

This suggests a trade-off between code structure and energy saving when writing the code. That means, in certain cases, we could inline some thin and highly-invoked methods in the code, at the cost of losing the integrity of the structure of the code to some extent.

Only one arithmetic operation, namely *Multi\_float\_float*, is a member of the top 10, and there are only six arithmetic operations in the top 30. They together cost only 6.1% of the overall energy consumption of the application, which is somewhat unexpected.

Later in block-level energy accounting, we will see that assignments, comparisons and *Array Reference* play significant roles in the overall energy consumption. This is not only because they are frequently used, but also because they are costly as operations themselves, as shown in Figure 1.

*Block Goto* operations are expensive as well. Based on the types of conditionals and loops where "Block Goto" occurs, they are classified into *BlockGoto\_if*, *BlockGoto\_for* and *BlockGoto\_while*. The result shows that they cost different amounts of energy as operations themselves, respectively 6.7  $\mu$ J, 4.1  $\mu$ J and 1.1  $\mu$ J. Together with *Method Invocation*, they take up 37.6% of the total energy consumption of the application.

#### Block Level.

In the execution cases, we have 108 active blocks with a wide diversity of energy usage. In Figure ??, "In Application" means running the Click & Move scenario with the full set of blocks (that is, ignoring the execution cases described in Section 3.3 in which some blocks are removed). The total cost of a block "In Application" is plotted as an orange bar, reflecting both its cost and the number of times it is executed. The cost of a fixed number (3000) of executions of one block are calculated by multiplying its single-execution cost by 3000. This helps us to compare the single-execution costs of different blocks. The costs of blocks at "3000-Times-Execution" are plotted as green bars.

Similar to energy distribution on operations, only a small number (11 blocks) of all the blocks use up nearly half of the entire cost, which indicates that putting efforts on optimizing a small group of blocks can achieve significant energy saving.

There are two factors that make one block costly "In Application". The first factor is a large number of executions.

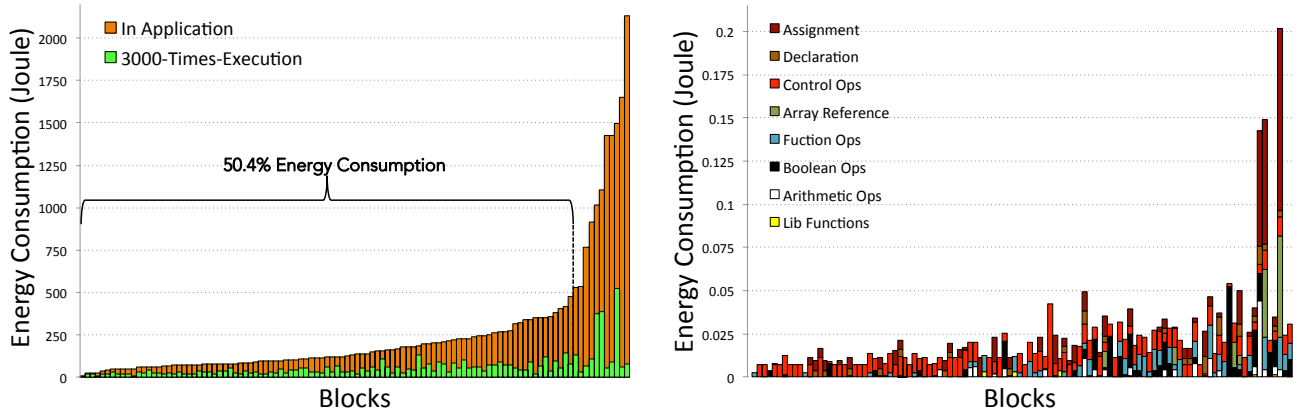


Figure 2: Energy distribution in Click & Move. Blocks are sorted by the order of their run-time energy costs In Application.

For example, the most costly block "In Application" (the rightmost orange bar in Figure ??) has a large number of execution times. This block takes only  $30.6 \mu\text{J}$  for single-execution but 2.1 Joules when running "In Application". The second factor is the energy consumption of the block itself. For example, we can see three prominent green bars in Figure ??, whose single-execution costs are  $201.5 \mu\text{J}$ ,  $146.9 \mu\text{J}$  and  $142.8 \mu\text{J}$ . We will later zoom in these three blocks to see which operations contribute to their energy costs.

We can further observe the energy proportions of operations in each block in Figure ?. To illustrate, operations are grouped into eight classes. Specifically, the "Block Goto" operations and *Method Invocation* are gathered in *Control Ops*; the parameter passing and the value returns of methods are in *Function Ops*; the comparisons and Booleans are in *Boolean Ops*; all the arithmetic computations are in *Arithmetic Ops*; all the library functions are in *Lib Functions*.

Most of the blocks cost less than  $25 \mu\text{J}$  for single-execution. In these blocks, *Control Ops* occupy the major part of the energy consumption, in contrast, *Arithmetic Ops* only take a tiny proportion.

For those three most prominent blocks, assignments and *Array Reference* are the biggest energy consumers. Furthermore one of the three blocks has the largest proportion of *Arithmetic Ops* among all the blocks.

The most expensive block "In Application" consists of three even parts: *Control Ops*, *Function Ops* and *Boolean Ops*. This block is the main entrance of the game engine to draw and display frames, so its work is dominated by conditional judgments and method invocations.

## 5.2 Code Optimization

The most important consideration of app developers is to guarantee the correctness of software, which should then be followed by energy efficiency. So our energy-aware programming approach is adopted at the end of software engineering life circle when the software system is in general

Table 3: The top 10 most costly blocks in Click & Move.

Block ID	Energy Cost (mJ)
CCNode.visit()	2128.6
CCNode.transform()	1648.4
CCTextureAtlas.putVertex()	1494.4
CCNode.visit().if_4.for_1	1426.8
CCNode.transform().if_1	1426.3
CCTextureAtlas.putTexCoords()	1107.8
CCAtlas.updateValues().for_1	1018.7
CCNode.visit().if_3.for_1	915.7
CCSprite.draw()	766.9
CCTexture2D.name()	537.5

complete.

We look into the top 10 costly blocks "In Application" (see Table 3). For example, *CCNode.visit()* is the entrance block of the *visit()* function; *CCNode.visit().if\_4.for\_1* is the body block of the *for* loop. These 10 blocks are distributed in seven methods, so the code review is straightforward. We find four easy opportunities to improve energy efficiency of some blocks: *CCNode.visit()*, *CCNode.visit().if\_4.for\_1* and *CCTexture2D.name()*. There are also other opportunities in other blocks supposed possible to save energy, but requiring more efforts and gaining little. For example, *CCAtlas.updateValues().for\_1* has several busy arithmetic expressions. Usually it is supposed that replacing the busy expression with a variable would reduce energy, however in this case the overhead of variable declaration counteracts the saved energy.

The four opportunities to reform the code are very simple and effective, but can only be discovered by the operation-level information. The changes are shown as follows:

### If Combination.

This change is made in the most costly block *CCNode.visit()*, which has two comparisons, two Boolean operations, one

---

**Program 1** Simplified parts of **original** code in *CCNode.visit()*

---

```

9
    if (children_ != null) {
        if_body1;
    }
    draw(gl);
    if (children_ != null) {
        if_body2;
    }

```

---



---

**Program 2** The changed Program 1

---

```

9
    if (children_ != null) {
        if_body1;
        draw(gl);
        if_body2;
    } else {draw(gl);}

```

---

*Method Invocation* and one parameter passing. In fact, the two if headers make the same comparison, as shown in Program 1. We change the code to Program 2, which combines the two if statements and meanwhile keep it logically consistent with Program 1. By these means each execution of the block can reduce one comparison, and when the condition is false, it can additionally reduce one *BlockGoto\_if*.

---

**Program 3** Simplified parts of **original** code in *CCNode* class

---

```

9
    public void visit(GL10 gl) {
        .....
        transform(gl);
        .....
    }
    public void transform(GL10 gl) {
        tranform_body;
    }

```

---

*Inner-Class Method Inline.*

When "In Application", the *transform()* function is invoked 18903 times and mostly by the *visit()* function. We change the Program 3 to Program 4 by switching the body of *transform()* to the function call of *transform()* in *visit()*, meanwhile remaining the original definition of *transform()* in case that other parts of the code call it. This change can greatly decrease the number of calls to *transform()*s and thus *Method Invocations* that are costly. However, it may be at the cost of losing readability of the code (which might be partly compensated by adding explanatory comments).

*Loop-Invariant Code Motion.*

*CCNode.visit().if\_3.for\_1* and *CCNode.visit().if\_4.for\_1* are entrance blocks of the two for loops as seen in Program 5.

---

**Program 4** The changed Program 3

---

```

9
    public void visit(GL10 gl) {
        .....
        transform_body;
        .....
    }
    public void transform(GL10 gl) {
        transform_body;
    }

```

---

These two loops share a quantity, *children\_.size()*, which is computed in each iteration but actually constant. We thus hoist it outside the loop, as shown in Program 6, which vastly saves the energy of invoking and executing the *size()* function during every iteration. Meantime, we move the declaration of the *child* outside the loop, considering the cost of *Declaration\_Object* is about 2.97  $\mu$ J and also in the top 30.

*Inter-Class Method Inline.*

*CCTexture2D.name()* is the 10th most costly block and costs 537.5 mJ "In Application". However, its job is to simply get the value of the private member variable, *\_name*, of the class *CCTexture2D*. This method has only two callers in the code. So we consider to make this variable public and let the two callers directly get access to the variable, which avoids the cost of *Method Invocation*. This change may harm the encapsulation of data, however, only one member of one class is changed. The trade-off between energy-saving and data encapsulation will be at last decided by developers.

---

**Program 5** The full version of Program 2

---

```

9
    if (children_ != null) {
        for (int i=0; i<children_.size(); ++i) {
            CCNode child = children_.get(i);
            if (child.zOrder_ < 0) {
                child.visit(gl);
            } else
                break;
        }
        draw(gl);
        for (int i=0; i<children_.size(); ++i) {
            CCNode child = children_.get(i);
            if (child.zOrder_ >= 0) {
                child.visit(gl);
            }
        }
    } else {draw(gl);}

```

---

## 5.3 Evaluation

Figure 3 illustrates the energy dissipation of the software without and with the changes introduced in the previous section. From left to right, the bars indicate cumulative effects of the changes. For example, "+ *If Comm*" is the en-



---

**Program 6** The changed Program 5

---

9

```
CCNode child = new CCNode(); //added
int children_size = children_.size(); //added
if (children_ != null) {
for (int i=0; i<children_size; ++i) { //changed
child = children_.get(i); //changed
if (child.zOrder_ < 0) {
child.visit(gl);
} else
break;
}
draw(gl);
for (int i=0; i<children_size; ++i) { //changed
child = children_.get(i); //changed
if (child.zOrder_ >= 0) {
child.visit(gl);
}
}
} else {draw(gl);}
```

---

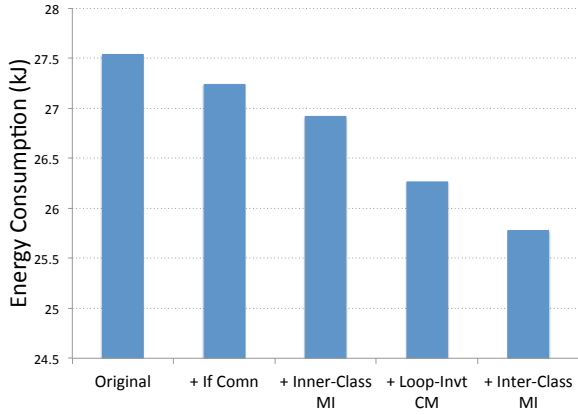


Figure 3: Energy consumption of the code without and with the changes in Click & Move.

ergy consumption of the original code with the change of "If Combination"; "+ Inner-Class MI" is the energy consumption of the code with the changes of both "If Combination" and "Inner-Class Method Inline". In total, these four simple changes save 6.4% of the entire energy consumption without influencing the functionality of code. These changes are made in the basic part of the game engine, which most applications will be bases on, so any gain here can have fundamental impact. Furthermore, these changes are made with little knowledge about the algorithm of the code, the developers who designed the code are surely able to improve the code much more and achieve far more energy-saving, if the energy model was available to them.

## 6. THE ORBIT SCENARIO

In this section, we briefly describe the energy accounting for the Orbit scenario. Then we improve the most costly blocks focusing on the expensive operations. In Section 6.3,

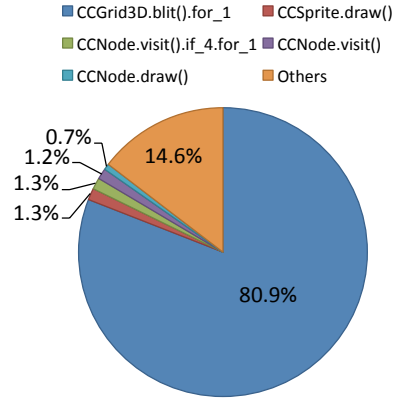


Figure 4: The energy proportions of blocks "In Application" in the Orbit scenario

the experimental result shows that the improvement can save 50.2% of the overall energy consumption.

### 6.1 Energy Accounting

In the Orbit scenario, the block *CCGrid3d.blit().for\_1* dominates the overall energy consumption. As shown in Figure 4, 80.9% of the entire cost is consumed by this block. The second most costly block consumes only 1.3%. "In Application" here means running the Orbit scenario without removing any block. Later in Section 6.2, we only focus on this single block.

### 6.2 Code Optimization

Program 7 shows the original code of *CCGrid3D.blit().for\_1*. In this block, the *Control Ops (BlockGoto\_for* and *Field Reference)* use up 35.6% of the energy; *Boolean Ops* use up 20.5%; the assignments use up 16.7%; *Arithmetic Ops* use up 14.0%; *Lib Functions* use up 13.3%. We find three easy changes to reduce or replace the pricey operations.

#### *Loop-Invariant Code Motion.*

In this block, the value of *vertices.limit()* is the constant 2112; we therefore hoist it outside the loop and replace it with the variable *limit*, as shown in Program 8. This change avoids invocations and executions of *vertices.limit()* and at the same time decreases a small amount of *Field Reference*.

---

**Program 7** The original code of *CCGrid3D.blit().for\_1*

---

9

```
for (int i = 0; i < vertices.limit(); i=i+3) {
mVertexBuffer.put(vertices.get(i));
mVertexBuffer.put(vertices.get(i+1));
mVertexBuffer.put(vertices.get(i+2));
}
```

---

---

**Program 8** The changed Program 7

---

```
9
int limit = vertices.limit(); //added
for (int i = 0; i < limit; i=i+24) { //changed
mVertexBuffer.put(vertices.get(i));
mVertexBuffer.put(vertices.get(i+1));
mVertexBuffer.put(vertices.get(i+2));
...
mVertexBuffer.put(vertices.get(i+23)); //added
}
```

---

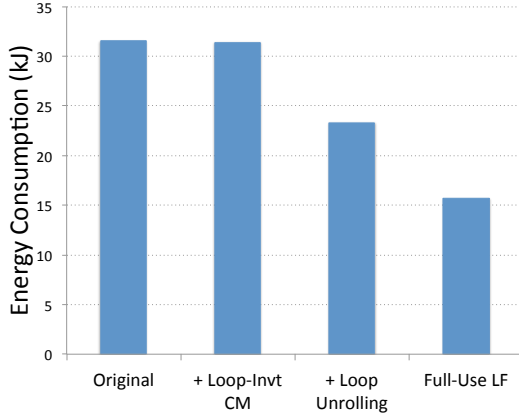


Figure 5: Energy consumption of the code without and with the changes in Orbit.

#### Loop Unrolling.

Also as shown in Program 8, we duplicate the loop body eight times, reducing the times of comparisons, *BlockGoto\_fors*, assignments and additions. Note that we set the value of the increment as 24 since 24 is a factor of the *limit*, 2112.

#### Full Use of Library Function.

The job of Program 7 or Program 8 is to get all the elements in *vertices* one by one and put them one by one into *mVertexBuffer*. Program 7 can be simply replaced by one line: *mVertexBuffer.put(vertices.asReadOnlyBuffer())*. This puts all the elements of *vertices* into *mVertexBuffer*. This change realizes the same functionality using the already existing library function, which is one of the key library functions already compiled into native code.

### 6.3 Evaluation

Figure 5 shows the cumulative effects of the code changes on energy consumption. In contrast to the other columns, "Full-Use LF" does not take previous changes into account and means only replacing Program 7 with the built-in library function as stated above. The figure shows that loop-invariant code motion does not gain much energy saving because *vertices.limit()* is a library function and in addition uses a very small percentage of energy consumption. On the other hand, loop unrolling achieves 25.8% energy saving due to the reduction of the amount of *Control Ops*, compar-

isons and assignments, which occupy most of the cost. The most effective change is the replacement to the library function, avoiding the waste of 50.2% energy use because this library function has been compiled into native code before execution, in contrast the Java source code need run-time interpretation which of course incurs an energy cost. The result implies that it is a good idea for developers to make a good use of library functions rather than implementing the same function with Java source code. The discovery of this source of inefficiency was assisted by the energy accounting.

## 7. THE WAVES SCENARIO

In this section, similarly, we first analyze the energy features of the blocks in the Waves scenario, based on which we modify the code and then evaluate the effects of changes on energy consumption.

### 7.1 Energy Accounting

Unlike the Orbit scenario where only one block dominates energy cost, in the Waves scenario the costs of the top eight blocks are at the same order of magnitude of kJ, as listed in Table 4. The *CCGrid3D.blit().for\_1* is also employed in this scenario and is the most costly as well among all the blocks. The majority of blocks in Table 4 are directly or indirectly invoked by *CCWaves3D.update().for\_1*. *for\_1*, as shown in Program 9. The purpose of these methods is mainly to set or get the values of member variables, so a large part of energy consumption goes to assignments, *Function Ops* and *Control Ops*. It was not expected that the code spends such a large amount of energy on simple set and get functions.

### 7.2 Code Optimization

#### Full-Use of Library Function.

We mentioned previously in Section 6.2 the optimization for *CCGrid3D.blit().for\_1* where we replace the entire Program 7 with one line of code making use of library functions. We keep this change in this scenario. For other blocks, we come up with one modification as below.

---

**Program 9** The original code in *CCWaves3D.update()*

---

```
9
int i, j;
for( i = 0; i < (gridSize.x+1); i++ ) {
for( j = 0; j < (gridSize.y+1); j++ ) {
CCVertex3D v=originalVertex(ccGridSize.ccg(i,j));
...
setVertex(ccGridSize.ccg(i,j), v);
}
}
```

---

#### Method Inline & Code Motion.



Table 4: Top 10 most costly blocks "In Application" in the Waves scenario and the energy percentages of different kinds of operations in each block.

Block ID	#Executions	Energy Cost (mJ)	Assi.	Decl.	Cont.	Func.	Bool.	Arit.	Libr.
CCGrid3D.blit().for_1	112193	8094.1	16.7%	0%	35.6%	0%	20.5%	14.0%	13.3%
CCVertex3D.CCVertex3D()	40219	5232.0	27.2%	0%	10.0%	62.8%	0%	0%	0%
CCWaves3D.update().for_1.for_1	34604	4088.7	10.7%	0%	32.1%	0%	14.7%	39.0%	2.2%
ccGridSize.ccg()	42275	3769.1	0%	0%	32.1%	67.9%	0%	0%	0%
CCGrid3DAction.setVertex()	31856	3285.4	14.6%	7.8%	30.9%	46.7%	0%	0%	0%
CCGrid3DAction.originalVertex()	36566	2891.3	19.1%	10.2%	40.3%	30.4%	0%	0%	0%
CCNode.getGrid()	49119	2145.1	0%	0%	58.1%	41.9%	0%	0%	0%
ccGridSize.ccGridSize()	10570	1173.8	30.3%	0%	31.6%	38.0%	0%	0%	0%
CCGrid3D.setVertex()	3944	657.2	10.1%	1.6%	32.8%	28.9%	0%	26.4%	0.2%
CCGrid3D.originalVertex()	2785	374.2	14.0%	1.9%	33.4%	17.9%	0%	32.8%	0%

**Program 10** Program 9 after Method Inline & Code Motion

```

9
ccGridSize ccgridsize = new ccGridSize(0,0); //added
CCGrid3D ccgrid3d =
(CCGrid3D) target.getGrid(); //added
CCVertex3D v = new CCVertex3D(0,0,0); //added
int i, j;
for( i = 0; i < (gridSize.x+1); i++ ) {
for( j = 0; j < (gridSize.y+1); j++ ) {
ccgridsize.x=i;ccgridsize.y=j; //added
v =ccgrid3d.originalVertex(ccgridsize); //changed
...
ccgrid3d.setVertex(ccgridsize, v); //changed
}
}

```

As shown in Program 9, the three functions called in the inner loop body are *CCGrid3DAction.originalVertex()*, *ccGridSize.ccg()* and *CCGrid3DAction.setVertex()*, which respectively cost 2891.3 mJ, 3769.1 mJ and 3285.4 mJ "In Application". Note that, *CCGrid3DAction* is the parent class of *CCWaves3D*, so in Program 9 *originalVertex()* and *setVertex()* can be directly called without referring to their class names. As seen in Program 10, we unpack these three methods in this block: the first and fourth "added" lines are unpacked *ccGridSize.ccg()*; the second "added" and first "changed" lines are unpacked *CCGrid3DAction.originalVertex()*; the second "added" and second "changed" lines are unpacked *CCGrid3DAction.setVertex()*. This change removes all the *Method Invocations*, parameter passing and value returns related to these three functions invoked by this block. Note that the first three "added" lines are located outside the loop in order to reduce energy consumption of the process of initializing objects and calling *CCNode.getGrid()*.

### 7.3 Evaluation

Figure 6 shows the cumulative effects of changes on energy consumption of CPU and GPU (note that previous figures only showed the CPU energy consumption because the GPU energy consumption did not vary noticeably), and the

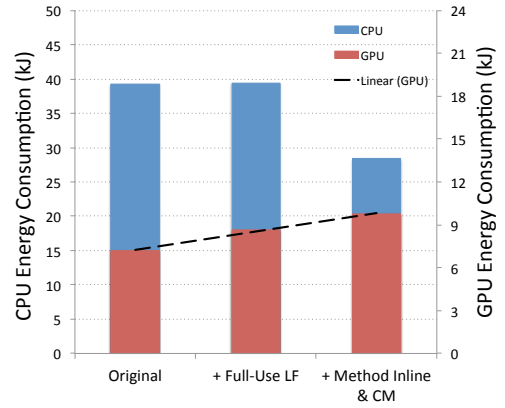


Figure 6: CPU and GPU Energy consumption of the code without and with the changes in Waves.

dashed line indicates the linear trend of the GPU energy consumption. In the case of games, the target frame rate is usually 60 Hz; when the game overloads the CPU the rate will decrease, and when the workload is light, even very light, the rate is generally fixed to 60 Hz. The rate in "Original" is around 36 Hz; that in "+ Full-Use LF" is around 50 Hz; that in "+ Method Inline & CM" is around 60 Hz. The change of *Full-Use LF* (full use of library function) does not save energy in the CPU because the execution of the original Waves actually overloads the CPU capacity, so the improvement of code enables the device to generate more frames every second. Consequently, the CPU does the same volume of work and consumes the equal amount of energy, the GPU does more work and consumes more energy, as seen in Figure 6. After this change, when we apply the method inline and code motion, 27.7% of the overall CPU energy is saved, and for the same reason the GPU consumes slightly more. This experimental result indicates that our approach not only saves energy but also potentially boosts performance, which benefits the user doubly.

## 8. RELATED WORK

### Energy Modeling.

From the hardware side, research on energy modeling have been done at the circuit level (see the survey [28]), gate level [26, 27] and register-transfer level [15]. Later, research focus shifted towards high-level modelings, such as software and behavioral levels [25].

Energy modeling techniques for software start with the basic instruction level, which calculates the sum of energy consumption of basic instructions and transition overheads [8, 42]. Gang et al. [36] base the model at the function-level while considering the effects of cache misses and pipeline stalls on functions. T. K. Tan et al. [41] utilize regression analysis for high-level software energy modeling.

However, the run-time context considered in the above works is unsophisticated, free from user inputs, a virtual machine, dynamic compilation, and etc. Furthermore the software stack below the level that they deal with (such as the level of the basic or assembly instruction) is relatively thin.

When research is focused on the energy use of mobile applications, the level of granularity of the techniques is increased as well. An important part of such efforts is the use of operating system and hardware features as predictors to estimate the energy consumption at the component, virtual machine and application level [11, 18, 33, 37, 45, 46].

Shuai et al. [14] and Ding et al. [20] propose approaches to obtain source line energy information. The former requires the specific energy profile of the target system, and the workload is fine-tuned. The latter utilizes advanced measurement techniques to obtain the source line energy cost.

Compared with approaches above, our latest work explores the idea of identifying energy operations and constructing a fine-grained model based on operations which is able to capture energy information at a level more fine-grained than source line.

### Energy-Saving Techniques.

A large amount of research effort on energy-saving for mobile devices has been focused on the main hardware components, such as the CPU, display and network interface. The CPU-related techniques involve dynamic voltage and frequency scaling [?] and heterogeneous architecture [13, 23]. Techniques targeting the display include dynamic backlight dimming [9, 32] and tone-mapping based back light scaling [5, 16]. Network-related techniques try to exploit idle and deep sleep opportunities [24, 39], shape the traffic patterns [10, 35], and so on. Such work attempts to reduce energy dissipation by optimizing the hardware usage; on the other hand, several pieces of work aim at designing new hardware and devices [?, ?].

There is a significant research focus on software optimization for saving energy. The basic work seeks to understand how the different methods, algorithms and design patterns of software influence the energy consumption. For example, [?, ?, ?] propose new routing techniques and protocols that are aware of energy consumption, which are evaluated

by comparing with traditional techniques. For another example, [?] investigates the affects of different sorting algorithms on the energy consumption with respect to the algorithm's input-size.

Considering design patterns, Litke et al. [?] conduct an experiment showing how big the difference of energy consumption is, before and after the application of design patterns, such as *factory method pattern*, *observer pattern*, and etc. The result reveals that except for one example the use of design patterns does not increase the energy use noticeably. Comparable work to [?] is done by [?]; they explores more design patterns and arrive at the conclusion that applying design patterns can both increase and decrease energy dissipation, so design-level artifacts cannot be used to estimate the impacts of design patterns on energy use.

Vetrò et al. [43] define the concept of "energy code smells" that are the code patterns (such as self assignment, repeated conditionals and useless control flow) that suggest energy inefficiency. However, the code patterns selected in [43] have very little influence (less than 1.0%) on energy consumption.

Regarding code refactoring for energy-saving, Ding et al. [19] perform a small scale evaluation of several commonly suggested programming practices that may reduce energy. Its result shows that reading array length, accessing class field and method invocation all cost noticeable energy. However, this work only provides a small number of tips to developers on how to make the code more energy-efficient.

To the best of our knowledge, the state of art before this paper did not connect the understanding of energy consumption with refactoring for such a high-level source code as Java. Our research proposes an energy-aware programming approach, which is guided by the operation-based source-level energy model. The experimental evaluation demonstrates that the approach is an effective and practical approach to energy-aware mobile application development.

## 9. CONCLUSION

In this paper, we propose an energy-aware programming approach for mobile app development, guided by an operation-based source-level energy model. The approach consists of 1) construction of an operation-based energy model by mining the data generated in a range of well-designed execution cases; 2) capturing energy characteristics of the code based on the model; 3) improving the code by removing, reducing or replacing the expensive operations in the costly blocks.

We evaluate this approach on a physical Android development board with two ARM quad-core CPUs and on a real-world game engine. In this case study our approach has a significantly positive impact on energy-saving. For different scenarios, this approach can save energy between 6.4% and 50.2%. The findings also indicate that the performance of code is a potential by-product of this approach, which improves the user experience more.

## 10. REFERENCES

- [1] *Android Debug Bridge*.  
<http://developer.android.com/tools/help/adb.html>.
- [2] *Cocos2d-Android*.  
<https://code.google.com/p/cocos2d-android/>.
- [3] *Dalvik Virtual Machine*.  
<http://source.android.com/devices/tech/dalvik/>.
- [4] *Report: U.S. Smartphone Penetration Now At 75 Percent*. <http://marketingland.com/report-us-smartphone-penetration-now-75-percent-117746>, 2015.
- [5] B. Anand, K. Thirugnanam, J. Sebastian, P. G. Kannan, A. L. Ananda, M. C. Chan, and R. K. Balan. Adaptive display power management for mobile games. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 57–70, New York, NY, USA, 2011. ACM.
- [6] *Android. A JIT Compiler for Android's Dalvik VM*.  
<http://www.android-app-developer.co.uk/android-app-development-docs/android-jit-compiler-androids-dalvik-vm.pdf>.
- [7] D. Bogdanas and G. Roşu. K-java: A complete semantics of java. *SIGPLAN Not.*, 50(1):445–456, Jan. 2015.
- [8] C. Brandolese, W. Fomacian, F. Salice, and D. Sciuto. An instruction-level functionality-based energy estimation model for 32-bits microprocessors. In *Design Automation Conference, 2000. Proceedings 2000*, pages 346–350, 2000.
- [9] W.-C. Cheng and M. Pedram. Power minimization in a backlit ftled display by concurrent brightness and contrast scaling. *Consumer Electronics, IEEE Transactions on*, 50(1):25–32, Feb 2004.
- [10] C. Chiasserini and R. Rao. Improving battery performance by using traffic shaping techniques. *Selected Areas in Communications, IEEE Journal on*, 19(7):1385–1394, Jul 2001.
- [11] M. Dong and L. Zhong. Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 335–348, New York, NY, USA, 2011. ACM.
- [12] C. Edwards. Lack of software support marks the low power scorecard at dac. In *Electronics Weekly.*, pages 15–21, June 2011.
- [13] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Taylor. The greendroid mobile application processor: An architecture for silicon's dark future. *Micro, IEEE*, 31(2):86–95, March 2011.
- [14] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan. Estimating mobile application energy consumption using program analysis. In *Proceedings of the 2013 International Conference on Software Engineering, ICSE '13*, pages 92–101, Piscataway, NJ, USA, 2013. IEEE Press.
- [15] C.-T. Hsieh, Q. Wu, C.-S. Ding, and M. Pedram. Statistical sampling and regression analysis for rt-level power evaluation. In *Computer-Aided Design, 1996. ICCAD-96. Digest of Technical Papers., 1996 IEEE/ACM International Conference on*, pages 583–588, Nov 1996.
- [16] A. Iranli and M. Pedram. Dtm: Dynamic tone mapping for backlight scaling. In *Proceedings of the 42Nd Annual Design Automation Conference, DAC '05*, pages 612–617, New York, NY, USA, 2005. ACM.
- [17] X. Jiang, P. Dutta, D. Culler, and I. Stoica. Micro power meter for energy monitoring of wireless sensor networks at scale. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks, IPSN '07*, pages 186–195, New York, NY, USA, 2007. ACM.
- [18] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. A. Bhattacharya. Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC '10*, pages 39–50, New York, NY, USA, 2010. ACM.
- [19] D. Li and W. G. J. Halfond. An investigation into energy-saving programming practices for android smartphone app development. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software, GREENS 2014*, pages 46–53, New York, NY, USA, 2014. ACM.
- [20] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan. Calculating source line level energy information for android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013*, pages 78–89, New York, NY, USA, 2013. ACM.
- [21] X. Li and J. P. Gallagher. A top-to-bottom view: Energy analysis for mobile application source code. *CoRR*, abs/1510.04165, 2015.
- [22] X. Li, G. Yan, Y. Han, and X. Li. Smartcap: User experience-oriented power adaptation for smartphone's application processor. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE '13*, pages 57–60, San Jose, CA, USA, 2013. EDA Consortium.
- [23] F. X. Lin, Z. Wang, R. LiKamWa, and L. Zhong. Reflex: Using low-power processors in smartphones without knowing them. *SIGPLAN Not.*, 47(4):13–24, Mar. 2012.
- [24] J. Liu and L. Zhong. Micro power management of active 802.11 interfaces. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services, MobiSys '08*, pages 146–159, New York, NY, USA, 2008. ACM.

- [25] E. Macii, M. Pedram, and F. Somenzi. High-level power modeling, estimation, and optimization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(11):1061–1079, Nov 1998.
- [26] R. Marculescu, D. Marculescu, and M. Pedram. Adaptive models for input data compaction for power simulators. In *Design Automation Conference, 1997. Proceedings of the ASP-DAC '97 Asia and South Pacific*, pages 391–396, Jan 1997.
- [27] F. Najm. Transition density: a new measure of activity in digital circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 12(2):310–323, Feb 1993.
- [28] F. Najm. A survey of power estimation techniques in vlsi circuits. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):446–455, Dec 1994.
- [29] A. Ng. *CS229 lecture notes*. <http://cs229.stanford.edu/notes/cs229-notes1.pdf>, 2012.
- [30] Odroid. *Odroid-XUE*. <http://www.hardkernel.com/main/main.php>.
- [31] J. Pallister, S. Kerrison, J. Morse, and K. Eder. Data dependent energy modelling: A worst case perspective. *CoRR*, abs/1505.03374, 2015.
- [32] S. Pasricha, M. Luthra, S. Mohapatra, N. Dutt, and N. Venkatasubramanian. Dynamic backlight adaptation for low-power handheld devices. *IEEE Design Test of Computers*, 21(5):398–405, 2004.
- [33] A. Pathak, Y. C. Hu, and M. Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12*, pages 29–42, New York, NY, USA, 2012. ACM.
- [34] G. Pinto, F. Castor, and Y. D. Liu. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014*, pages 22–31, New York, NY, USA, 2014. ACM.
- [35] C. Poellabauer and K. Schwan. Energy-aware traffic shaping for wireless real-time applications. In *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*, pages 48–55, May 2004.
- [36] G. Qu, N. Kawabe, K. Usarni, and M. Potkonjak. Function-level power estimation methodology for microprocessors. In *Design Automation Conference, 2000. Proceedings 2000*, pages 810–813, 2000.
- [37] A. Shye, B. Scholbrock, and G. Memik. Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42*, pages 168–178, New York, NY, USA, 2009. ACM.
- [38] Soot. *A framework for analyzing and transforming Java and Android Applications*. <http://sable.github.io/soot/>.
- [39] J. Sorber, N. Banerjee, M. D. Corner, and S. Rollins. Turducken: Hierarchical power management for mobile devices. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services, MobiSys '05*, pages 261–274, New York, NY, USA, 2005. ACM.
- [40] STACKOVERFLOW. <http://stackoverflow.com>.
- [41] T. Tan, A. Raghunathan, G. Lakshminarayana, and N. Jha. High-level software energy macro-modeling. In *Design Automation Conference, 2001. Proceedings*, pages 605–610, 2001.
- [42] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):437–445, Dec 1994.
- [43] P. G. M. M. Vetro' A., Ardito L. Definition, implementation and validation of energy code smells: an exploratory study on an embedded system. In *ENERGY 2013 : The Third International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, pages 34–39, March 2013.
- [44] T. Šimunić, L. Benini, G. De Micheli, and M. Hans. Source code optimization and profiling of energy consumption in embedded systems. In *Proceedings of the 13th International Symposium on System Synthesis, ISSS '00*, pages 193–198, Washington, DC, USA, 2000. IEEE Computer Society.
- [45] C. Wang, F. Yan, Y. Guo, and X. Chen. Power estimation for mobile applications with profile-driven battery traces. In *Proceedings of the 2013 International Symposium on Low Power Electronics and Design, ISLPED '13*, pages 120–125, Piscataway, NJ, USA, 2013. IEEE Press.
- [46] L. Zhang, B. Tiwana, R. Dick, Z. Qian, Z. Mao, Z. Wang, and L. Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 105–114, Oct 2010.